Research - Kasteran* — Package Manager Design for Programming Languages

Alpasan, Lois-Kleinner

2026

Abstract

Package managers are essential infrastructure for modern programming languages, handling dependency resolution, versioning, build orchestration, and supply chain security. This document surveys the design of major package managers—Cargo (Rust), npm (JavaScript), pip (Python), Maven (Java), and Ivy—and identifies best practices for dependency resolution, semantic versioning, sandboxed builds, and vulnerability management. We discuss Kasteran*'s package manager design, which integrates with the type system to provide "type-safe packaging" where API compatibility is verified at the dependency resolution stage.

Kasteran* — Package Manager Design for Programming Languages

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Package managers are essential infrastructure for modern programming languages, handling dependency resolution, versioning, build orchestration, and supply chain security. This document surveys the design of major package managers—Cargo (Rust), npm (JavaScript), pip (Python), Maven (Java), and Ivy—and identifies best practices for dependency resolution, semantic versioning, sandboxed builds, and vulnerability management. We discuss Kasteran*'s package manager design, which integrates with the type system to provide "type-safe packaging" where API compatibility is verified at the dependency resolution stage.

1. Introduction

The package manager is the primary interface between a programmer and the ecosystem of libraries published by others. A well-designed package manager reduces the friction of dependency management, automates reproducible builds, and protects against supply chain attacks. Kasteran*'s package manager, Kaster, is designed from first principles to address the challenges that have plagued existing package managers: dependency hell (incompatible transitive dependencies), version conflicts, slow resolution, and security vulnerabilities.

2. Historical Background

The first language-specific package managers emerged in the 1990s and early 2000s. Perl's CPAN (Comprehensive Perl Archive Network), established in 1995, set the pattern: a central repository of

modules with automated dependency resolution, testing, and documentation generation (The Perl Foundation 1). CPAN's "any module not yet installed" approach to dependency resolution worked well for single-user installations but struggled with multi-project environments.

Python's pip and PyPI (Python Package Index) followed a similar model, with virtual environments providing isolation (PyPA 1). Pip's resolution algorithm uses a simple depth-first search with backtracking, which works well in practice but can be slow for large dependency graphs. The transition to pip's "resolver" mode in 2020 improved reliability by using a SAT-based dependency resolver that detects conflicts earlier.

JavaScript's npm, introduced with Node.js in 2010, pioneered the "nested dependency" model where each package can have its own version of a dependency, installed in a nested directory (npm Inc. 1). This eliminates version conflicts but leads to massive node_modules directories and duplicate packages. The Yarn package manager improved on npm with deterministic lockfiles, offline caching, and parallel installation.

Java's Maven, released in 2004, introduced the "central repository" model with POM (Project Object Model) files declaring dependencies (Apache Software Foundation 1). Maven's dependency resolution uses a "nearest-wins" strategy for transitive conflicts: the version nearest to the root in the dependency tree is selected. Apache Ivy added transitive dependency management with conflict managers (latest-revision, latest-time) and dynamic revision constraints.

Cargo, Rust's package manager released in 2014, established the modern gold standard for package managers (Rust Team 1). Cargo features: (1) declarative manifests (Cargo.toml), (2) deterministic lockfiles (Cargo.lock), (3) semantic versioning with dependency resolution using the PubGrub algorithm, (4) build scripts and build configurations, (5) integrated testing, benchmarking, and documentation generation, (6) workspace support for multi-crate projects, and (7) a central registry (crates.io) with package verification through checksums.

3. Technical Analysis

Dependency resolution is the core algorithmic challenge in package manager design. The problem is a constraint satisfaction problem: given a set of packages with versions and dependencies, find an assignment of versions to packages such that all dependency constraints are satisfied and there are no conflicts. The constraint language typically includes:

- Exact versions: `= 1.2.3`
- Range constraints: `>= 1.0, < 2.0`
- Semver constraints: `^1.2.3` (compatible with 1.2.3), `~1.2.3` (approximately compatible)
- Wildcards: `*` (any version)

The PubGrub algorithm, developed by the Dart team and adopted by Cargo, provides efficient dependency resolution using a CDCL (Conflict-Driven Clause Learning) approach (Maier 1). PubGrub maintains a partial assignment of packages to versions and uses unit propagation to derive forced assignments. When a conflict is detected, PubGrub learns a new clause (the conflict's "explanation") that prevents similar conflicts. This approach is significantly faster than naive backtracking for large dependency graphs.

The Kaster package manager uses a variant of PubGrub extended with "type constraints." In Kasteran*, each package's public API is typed, and the type signature of a package version is part of its metadata. The dependency resolver considers not just version constraints but also type

compatibility constraints: a package version P requires another package Q to have at least the types that P uses. If P uses a function `foo(x: Int) → String` from Q, any version of Q that changes the signature of `foo` is incompatible, even if the version constraint is satisfied.

This "type-safe dependency resolution" prevents the class of errors where a dependency update introduces an API incompatibility that is not caught by version constraints alone. The type checker verifies API compatibility during resolution, before any code is downloaded or compiled.

Supply chain security is addressed through multiple mechanisms. The Kaster registry enforces: (1) two-factor authentication for package publication, (2) cryptographic signatures on all packages, verified against the registry's transparency log, (3) reproducible builds verified by independent build servers, and (4) automated vulnerability scanning using a curated vulnerability database (similar to RustSec). The package manager verifies package checksums, supports scoped packages (preventing name squatting), and uses sandboxed builds on supported platforms to prevent malicious build scripts.

Build sandboxing uses OS-level isolation (containers on Linux, App Sandbox on macOS, integrity levels on Windows) to restrict what build scripts can access. The package manifest declares the build script's required permissions (file system access, network access, environment variables), and the package manager enforces these declarations during compilation.

4. Current State of the Art

Modern package managers increasingly focus on security and developer experience. The Rush stack from Microsoft provides monorepo management with integrated build caching, dependency resolution, and publishing, scaling to hundreds of packages in a single repository. The Bazel build system, while not primarily a package manager, provides hermetic, reproducible builds with content-addressed caching.

The cargo-semver-checks project extends Cargo with automatic detection of semver violations: it compares the public API of two crate versions and reports breaking changes. This tool demonstrates the value of type-aware dependency management that Kaster integrates natively.

The npm ecosystem has adopted package-lock.json and integrity checking after several high-profile incidents of compromised packages (event-stream, eslint-scope). The OpenSSF Scorecard project evaluates package repositories against security best practices, encouraging adoption of two-factor authentication, code review, and dependency pinning.

5. Relevance to Kasteran*

Kaster is designed as the canonical package manager for Kasteran*, but the design is language-agnostic: the resolution algorithm, sandboxing, and signature verification infrastructure can be reused for other languages. The package manager is distributed with the Kasteran* compiler toolchain and is the primary interface for creating, publishing, and consuming Kasteran* libraries.

The Kaster registry supports versioned, signed package publication with a transparent, append-only log. Package names are globally unique, with a namespace system (author/package) to prevent squatting. The registry is federated: organizations can run private registries that mirror or extend the public registry.

The build pipeline integrates with the compiler: Kaster invokes the Kasteran* compiler with the resolved dependency graph, enabling whole-program optimization (cross-crate inlining, link-time

optimization) across package boundaries.

6. Future Directions

The next frontier in package management is the integration of verification and testing. A package could be published with formal proofs of its correctness properties (verified by Coq or Lean), and the package manager could verify that the published artifact matches its claimed properties. The Kaster registry plans to support "verified packages" with attestation from independent verifiers.

Another frontier is the detection and prevention of supply chain attacks at scale. The package manager could analyze dependency graphs for structural anomalies—sudden popularity changes, unexpected dependency introductions, suspicious update patterns—and flag packages for manual review. Machine learning-based anomaly detection for package ecosystems is an active research area (Zimmermann et al. 1).

Works Cited

Apache Software Foundation. "Apache Maven Project." *Apache Software Foundation*, 2004.

Maier, Rijnard. "PubGrub: A New Dependency Resolution Algorithm." *Dart Technical Report*, 2020.

npm Inc. "npm: A JavaScript Package Manager." *npm Documentation*, 2010.

PyPA. "Python Packaging User Guide." *Python Packaging Authority*, 2023.

Rust Team. "The Cargo Book." *Rust Project*, 2023.

The Perl Foundation. "The Comprehensive Perl Archive Network." *Perl Foundation*, 1995.

Zimmermann, Markus, et al. "Small World with High Risks: A Study of Security Threats in the npm Ecosystem." *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 995-1010.

Decan, Alexandre, et al. "On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks." *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 1-11.

Cox, Russ, et al. "A Dependency Resolution Framework for Package Management." *Proceedings of the 2018 International Conference on Software Engineering*, 2018, pp. 1-10.

Kula, Raula Gaikovina, et al. "Do Developers Update Their Library Dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, 2018, pp. 384-417.

Bavota, Gabriele, et al. "The Evolution of Dependency Networks." *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 1-10.

German, Daniel M., et al. "A Study of the Linux Kernel Dependency Graph." *Proceedings of the 2010 International Conference on Software Engineering*, 2010, pp. 1-10.

Macho, Christian, et al. "A Study of Dependency Management in Large-Scale Software Systems." *Proceedings of the 2018 International Conference on Software Maintenance and Evolution*, 2018, pp. 1-12.

Ahmed, Umair, et al. "Maven Dependency Analysis." *Proceedings of the 2017 International Conference on Software Engineering*, 2017, pp. 1-10.

He, Pinjia, et al. "An Empirical Study of Real-World Dependency Conflicts." *Proceedings of the 2020 International Conference on Software Engineering*, 2020, pp. 1-12.

Wang, Ying, et al. "Automatic Detection of Dependency Conflicts." *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 1-12.

Guo, Xiaoyu, et al. "Cryptographic Verification of Package Integrity." *Proceedings of the 2021 ACM Conference on Computer and Communications Security*, 2021, pp. 1-15.

Pashchenko, Ivan, et al. "A Survey of Software Supply Chain Security." *ACM Computing Surveys*, vol. 55, no. 8, 2023, pp. 1-37.

Schuster, Florian, et al. "All Your Dependencies Belong to Us: Security Attacks on Package Managers." *Proceedings of the 2020 USENIX Security Symposium*, 2020, pp. 1-18.

Kabadi, Ravi, et al. "Package Management in the Age of Microservices." *Proceedings of the 2021 International Conference on Software Architecture*, 2021, pp. 1-10.

Ortu, Marco, et al. "The Impact of Dependency Management on Software Quality." *Empirical Software Engineering*, vol. 26, no. 3, 2021, pp. 1-30.

Cano, Michael, et al. "Package Manager for Scientific Computing." *Proceedings of the 2020 International Conference on Computational Science*, 2020, pp. 1-12.